Stacks

Prof. Dr. Murat GÖK

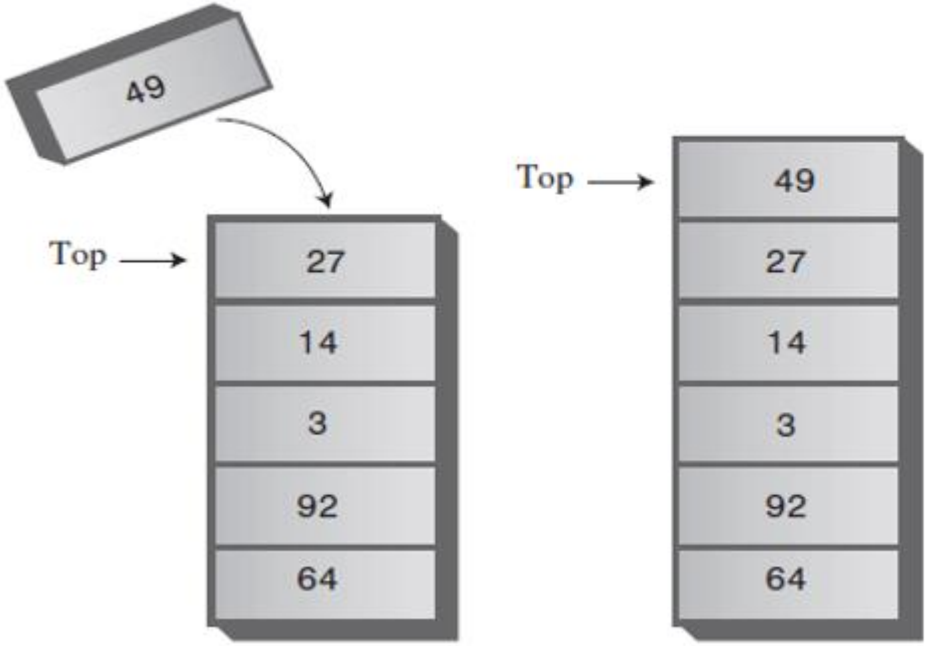
murat.gok@yalova.edu.tr

Stacks

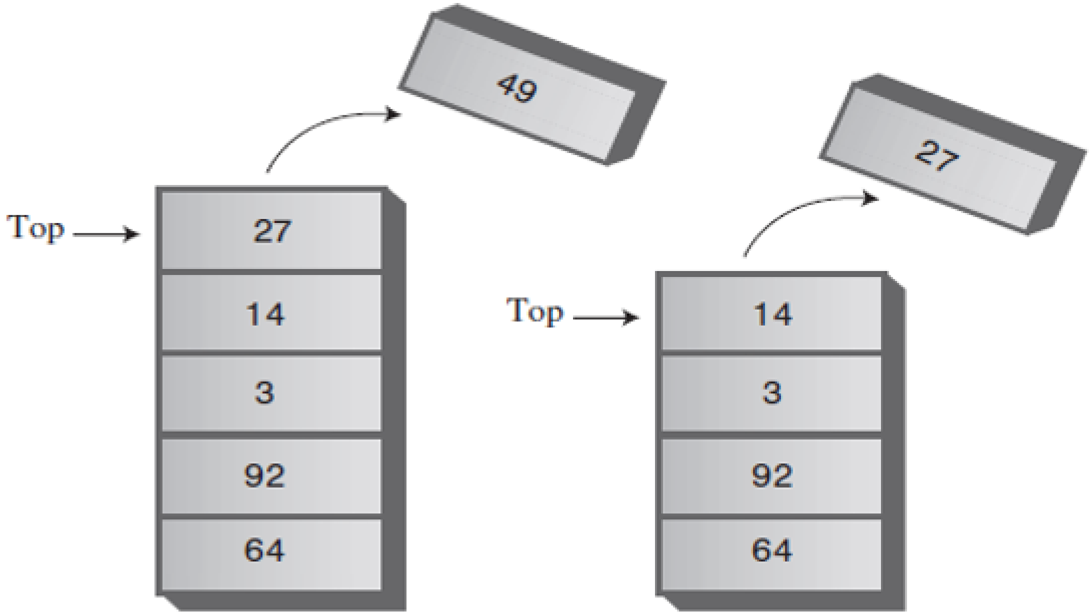
- A stack is said to be a Last-In-First-Out (LIFO) storage mechanism because the last item inserted is the first one to be removed.
- Most microprocessors use a stack-based architecture. When a method is called, its return address and arguments are pushed onto a stack, and when it returns, they’re popped off. The stack operations are built into the microprocessor.
- A stack is also a handy aid for algorithms applied to certain complex data structures. It used to help traverse the nodes of a tree.

Stacks

- Placing a data item on the top of the stack is called *pushing* it. Removing it from the top of the stack is called *popping* it. These are the primary stack operations.
- Trying to pop out an empty stack is called *underflow* and trying to push an element in a full stack is called *overflow*. Generally, we treat them as exceptions.



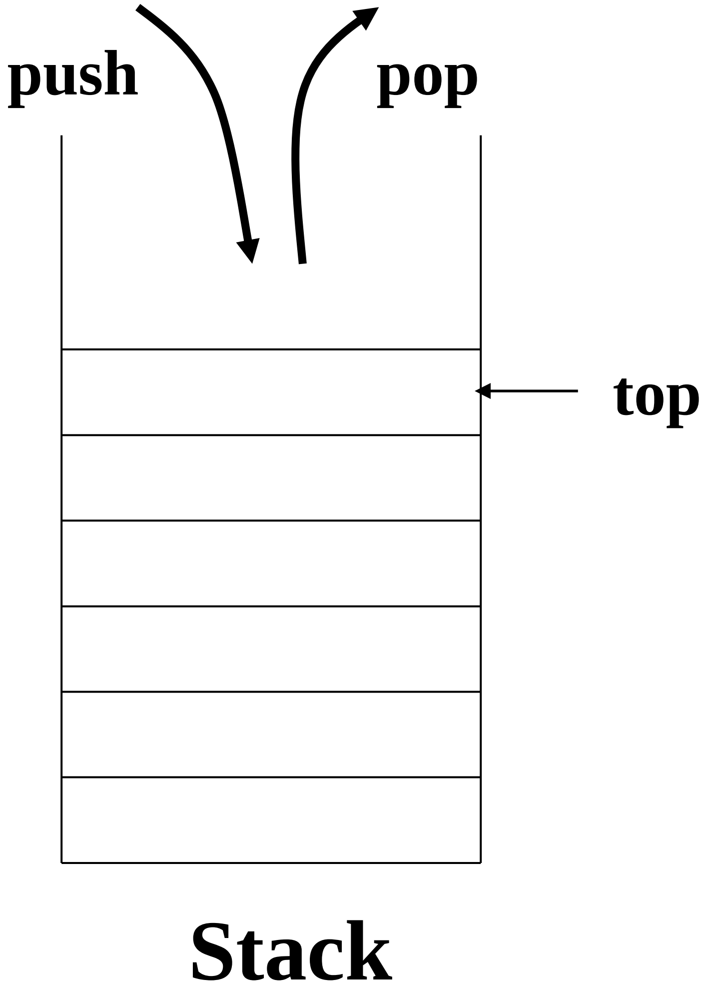
Pushing



Popping

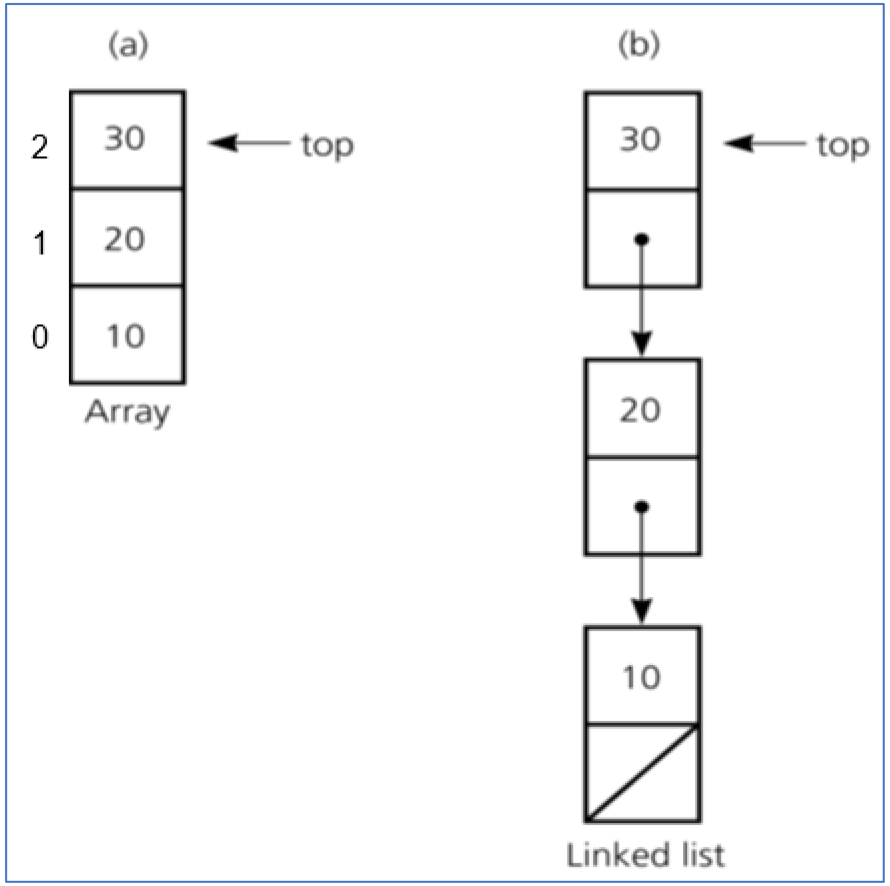
ADT Stack Operations

- **Main stack operations**
  - Push (int data): Inserts data onto stack.
  - int Pop(): Removes and returns the last inserted element from the stack.
- **Auxiliary stack operations**
  - int Top(): Returns the last inserted element without removing it.
  - int Size(): Returns the number of elements stored in the stack.
  - int IsEmptyStack(): Indicates whether any elements are stored in the stack or not.
  - int IsFullStack(): Indicates whether the stack is full or not.



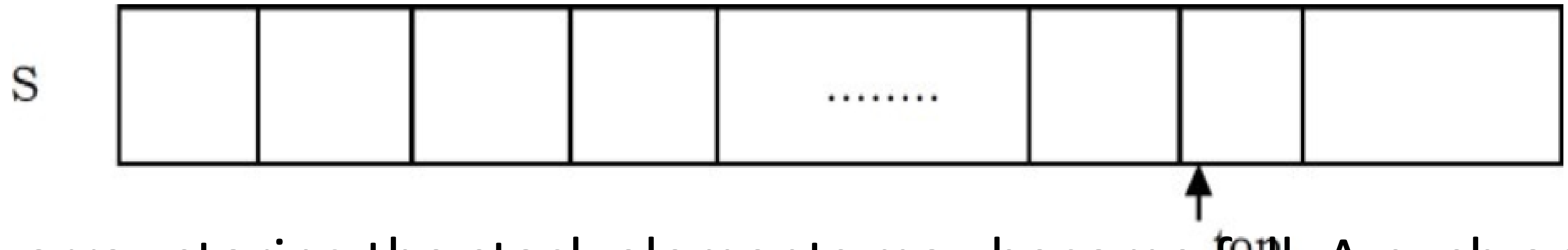
Implementations of the ADT Stack

- The ADT stack can be implemented using
  - An array
  - A linked list



Simple Array Implementation

- This implementation of stack ADT uses an array. In the array, we add elements from left to right and use a variable to keep track of the index of the top element.



- The array storing the stack elements may become full. A push operation will then throw a *full stack exception*.
- Similarly, if we try deleting an element from an empty stack it will throw *stack empty exception*.

Performance & Limitations

- **Performance**
- Let n be the number of elements in the stack. The complexities of stack operations with this representation can be given as:

Space Complexity (for n push operations)	O(*n*)
Time Complexity of Push()	O(1)
Time Complexity of Pop()	O(1)
Time Complexity of Size()	O(1)
Time Complexity of IsEmptyStack()	O(1)
Time Complexity of IsFullStackf)	O(1)
Time Complexity of DeleteStackQ	O(1)

- **Limitations**
- The maximum size of the stack must first be defined, and it cannot be changed. Trying to push a new element into a full stack causes an implementation-specific exception.

Example: Checking for Balanced Braces

- A stack can be used to verify whether a program contains balanced braces
- An example of balanced braces

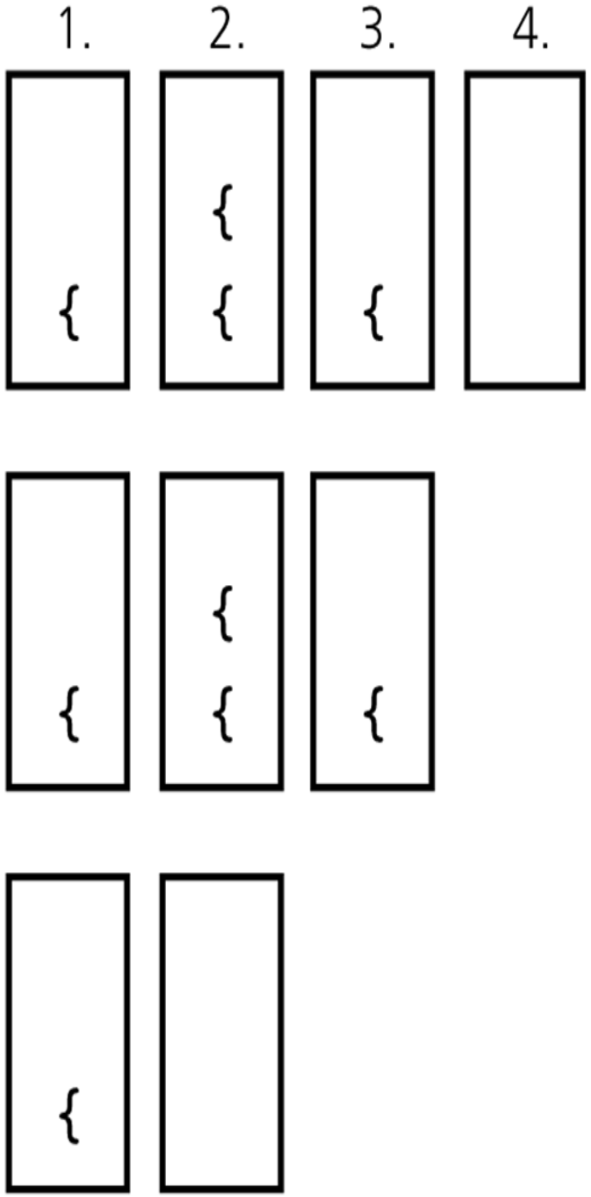
  abc{defg{ijk}{l{mn}}op}qr

- An example of unbalanced braces

  abc{def}}{ghij{kl}m

- Requirements for balanced braces
  - Each time we encounter a “}”, it matches an already encountered “{”
  - When we reach the end of the string, we have matched each “{”

Input string | Stack as algorithm executes



{a{b}c}

1. push "{"
2. push "{"
3. pop
4. pop

Stack empty ⟹ balanced

{a{bc}

1. push "{"
2. push "{"
3. pop

Stack not empty ⟹ not balanced

{ab}c}

1. push "{"
2. pop

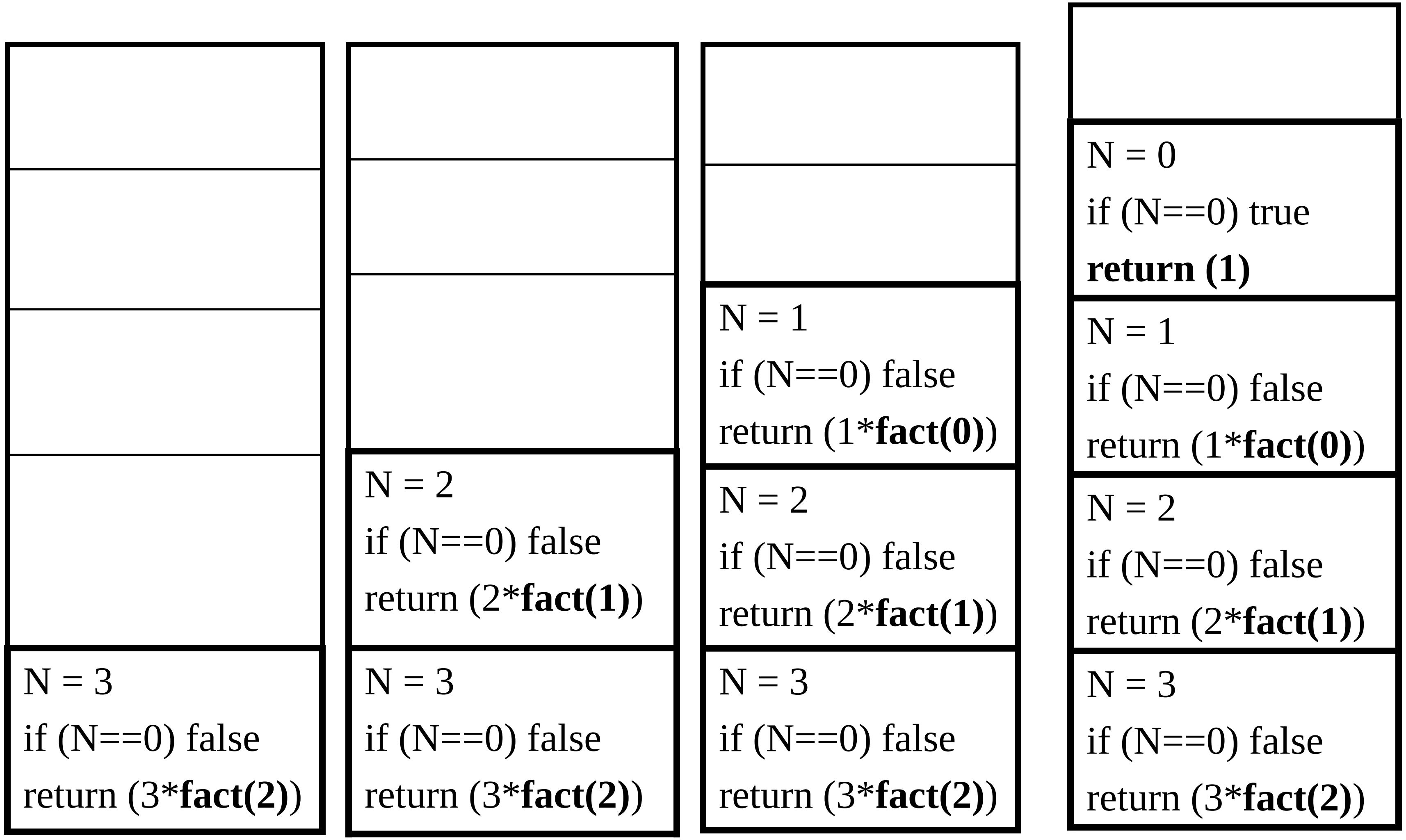
Stack empty when last "}" encountered ⟹ not balanced

Stacks - Example: Factorial function

- A strong relationship exists between recursion and stacks
  - Any recursive program can be rewritten as a nonrecursive program using stacks.
  - Here is the factorial code:

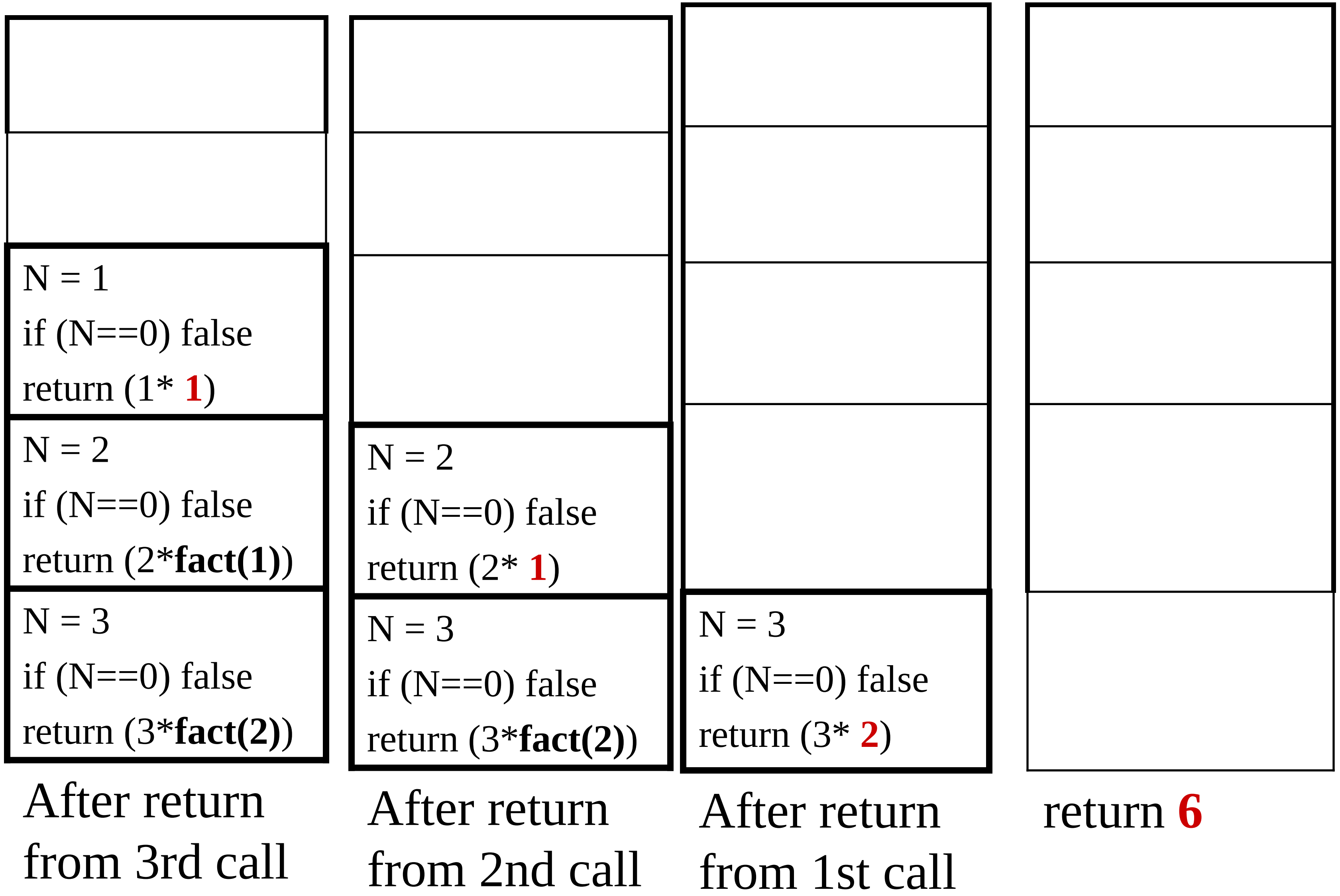
```
int fact(int n)
{
  if (n ==0)
        return (1);
  else
        return (n * fact(n-1));
}
```

Tracing the call `fact (3)`



After original call | After 1st call | After 2nd call | After 3rd call

Tracing the call `fact (3)`



Transforming a recursive factorial algorithm into a non-recursive one using an explicit stack:

code

Python

```python
def factorial_recursive(n):
    if n == 0:
        return 1
    else:
        return n * factorial_recursive(n - 1)


def factorial_non_recursive(n):
    stack = []
    result = 1

    while n > 0:
        stack.append(n)
        n -= 1

    while stack:
        result *= stack.pop()

    return result


# Example usage
n = 5
print(f"Recursive Factorial of {n}: {factorial_recursive(n)}")
print(f"Non-recursive Factorial of {n}: {factorial_non_recursive(n)}")
```


Thank you for
your attention